프로젝트#2

< 로봇우동가게 시뮬레이션 >

모바일 컴퓨팅

2013112130 컴퓨터공학과 정재엽

1. **주어진 코드 변수 분석**

Ta : 평균적으로 손님들이 들어오는 시간 간격

Ts : 평균적으로 손님들이 서비스 받는 시간

simultation_time : 시뮬레이션 시간

next_arrival_time : 다음 손님이 올 것이라고 예상되는 시간

next_departure_time : 다음 손님이 서비스를 받고 떠날 것이라고 예상되는 시간

elapsed_time : 시뮬레이션이 돌아간 총 시간

B : 시스템 안에 손님이 한명이라도 있는 시간(서비스를 하고 있는 시간)ㄴ

C : 서비스를 받고 떠난 손님의 총 수

s : 시스템 내에서 손님들이 머문 시간들의 총 합 (서비스를 받는 사람 + 기다리는 사람) = 현재 손님의 수 * (현재 손님이 도착한(떠난)시간 - 마지막 손님이 도착한(떠난시간))

L : s의평균값. 시스템 내의 손님들이 머무르는 시간 / 손님을 마지막으로 받은 시간까지의 합

tb : 손님이 0명일 경우, 손님이 1명 처음 들어왔을 때의 Base 시간

X : 시스템이 운영 중일 동안 서비스 한 수 (Throughput)

U : 시스템이 운영 중일 동안, 실제로 서비스를 한 시간 (Utilization)

L : 시스템 내 손님들이 시스템에 있었던 시간 평균

W : 시스템 내 손님들이 시스템에 있었던 시간 평균 / 시스템이 실제로 서비스를 한 수(X)

= 손님 한명당 시스템에 있던 시간

n : 시스템 내에 있는 손님의 수

Tn : 이전 손님이 떠난(도착한) 시간(마지막 이벤트 발생 시간

2. 결과 비교 분석

Ta = 200 : 평균 200초당 1명의 손님 ($\lambda = 1/200$ per second)

X = Ts = 100 : 1명의 손님이 서비스 받는 시간 ($\mu = 1/100$ per second)

L = Ls + Lw (시스템 내에서 서비스를 받고 있는 손님 및 대기중인 손님의 수)

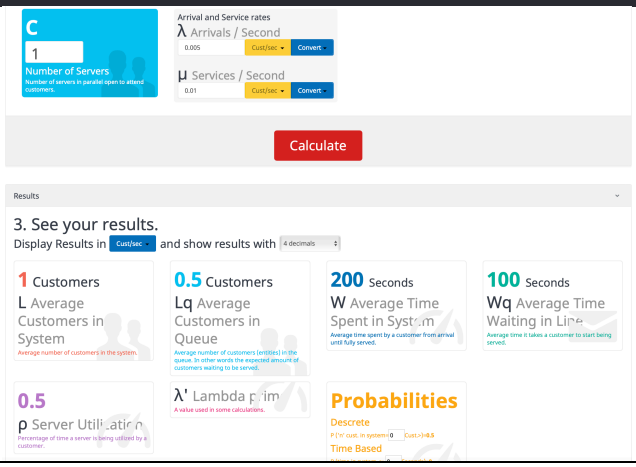
W = Ws + Wq (시스템에서 손님들이 머물고 있는 시간)

Utilization : 시뮬레이션 시간 동안 서버가 사용된 시간에 대한 비율

Throughput : 시뮬레이션 시간 동안 서비스를 받고 나간 손님에 대한 비율

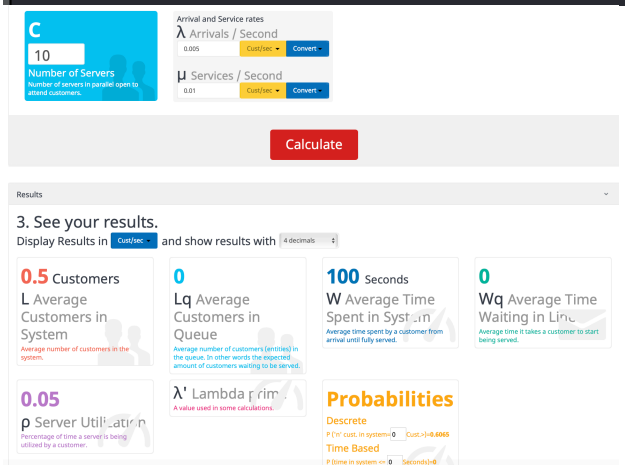
M/M/S=1/∞

```
throughput = 0.004749
utilization = 0.494459
mean customer no. in store = 0.971688
mean "Dwell" time per customer = 204.588604
L : 0.971688  W : 204.588604
Program ended with exit code: 0
```



M/M/S=10/∞

```
throughput = 0.005025
utilization = 0.369220
mean customer no. in store = 0.503896
mean "Dwell" time per customer = 100.282493
L : 0.503896  W : 100.282493
Program ended with exit code: 0
```



1. Throughput

M/M/S=1 < M/M/S=10 인 이유는 서버의 갯수가 더 많을 수록 손님을 더 많이 받을 수 있기 때문입니다.

2. Utilization

가게를 운영할 때 수익만큼 중요한 것이 회전율인데, 이 Utilization은 가게가 운영중인 동안 손님들이 가게에 있는 확률을 보여줍니다. S=10인 경우, 손님은 더 많이 받을 수 있지만 계속해서 가게에 손님들이 차고 있지 않기때문에, 임금 대비 수익률은 낮게 나올 것입니다. S=1인 경우, 총 서비스할 수 있는 손님의 수는 적지만 S=10과 Throughput이 비슷하여, 실질적으로 들어오는 수입은 비슷할 것입니다. 하지만 서비스하는 인원이 1명이기 때문에 임금 비는 1/10로, 더 효율적으로 가게를 운영할 수 있습니다.

3. Mean Customer # in store 또한 2번과 같이 가게의 수익에 중요한 역할을 끼칩니다. S=10인 경우, 가게에 와서 서비스를 받을 때 여러 명이 동시에 받기 때문에 가게 안에 머무르는 시간이 S=1 인 경우보다 적습니다.

4. Mean Dwell Time :

S=1인 경우 서비스를 받을 수 있는 사람이 1명이기 때문에 서비스를 받는 시간보다 서비스를 기다리는 시간이 더 길 것입니다. (이 부분은 구현하지 못하였기에, 파악하지 못하였습니다.) 하지만 S=10인 경우, 10명의 손님이 한 번에 서비스를 받을 수 있기 때문에 그만큼 가게 내에서 머무는 시간이 줄어들 게 됩니다.

5. 결과적으로 볼 때, 가게의 우동 로봇을 늘림으로서 원활한 서비스를 제공하고 있지만 가게의 측면에서는 낮은 효율을 보이게 됩니다.

3. 중요하게 다루어야 할 변수

해당 과제를 하면서 처음에는 next_departure_time을 중요하게 생각해야한다고 생각했습니다. 왜냐하면 과제를 진행하면서 이 값을 지정해주는 방법이 어려웠기 때문입니다. 하지만 3.을 작성하면서 우동 가게를 잘 운영하기 위해서는 Ta, 즉 손님들이 가게를 방문하는 시간 간격이 가장 중요하다고 생각하게 되었습니다. 왜냐하면 가게의 효율성을 높이기 위해서는 적절한 수의 로봇을 배치하고, 로봇의 수와 손님들이 기다리는 시간(Lq)의 적절한 compromising이 필요하기 때문입니다.

4. 추가 및 변경 변수

- M/M/S=10 모델을 디자인하면서,

기존의 Tn변수를 lastEventTime으로 Tb변수를 lastBusyTime으로 변경하여 사용하였습니다.

- M/M/S=10 모델에서는 M/M/S=1 모델과는 다르게 여러 개의 프로세스가 하나의 서버에서 서비스를 받을 수 있으며, 각각의 프로세스마다 서비스가 끝나는 시간이 각각 설정이 됩니다. 또한 서버가 현재 서비스하고 있는 프로세스보다 더 많은 프로세스를 수용할 수 없는 상태에서는 서비스 중인 프로세스 중에서 가장 빠른 Departure_Time을 갖는 프로세스를 기준으로 그 다음 프로세스의 Departure_Time이 설정이 됩니다. 따라서 서비스를 받는 프로세스들의 Departure_Time에 대한 정보를 알고 있어야합니다. 따라서 M/M/S=10 모델 과제에서 이를 관리하기 위하여

변수 :

int arrayIndex // custDeparture 배열에 저장되어있는 departure시간 인덱스

int nextDepartIndex // custDeparture에 인덱스를 사용한 뒤에 다음 departure시간 인덱스

double custDepartures[10] // 손님들의 떠나는 시간 관리를 위한 배열로 서버의 수 만큼 인덱스 생성

함수 :

int min_departure(double arr[], int capacity) // custDeparture에서 가장 빨리 떠나는 인덱스 반환

int idle_server(double arr[], int capacity) // custDeparture에서 사용가능한 인덱스 반환

를 추가하였습니다.

```
double expntl(double);
int min_departure(double arr[], int capacity);// 인덱스를 이용하여, 가장 빨리 떠나는 Customer를 가져오기 위한 함수
int idle_server(double arr[], int capacity);// 서버의 Capacity 확인
double Ta=200.0, Ts=100.0;
```

```
int min_departure(double arr[], int capacity)
{
    int index = 0;

    for (int i=1; i < capacity; i++)
    {
        if (arr[i] < arr[index])
            index = i;
    }
//    printf(" - INDEX = %d \n", index);
    return index;
}
```

```
int idle_server(double arr[], int capacity)
{
    int index = 0;
    bool idle = false;

    for (int i=1; (i < capacity && !idle); i++)
    {
        if (arr[i] == SIM_TIME)
        {
            index = i;
            idle = true;
        }
    }
//    printf(" - INDEX = %d \n", index);
    return index;
}
```

```
double lastBusyTime = 0.0;
int arrayIndex = 0;
int nextDepartIndex = 0;
double custDepartures[10];
// MM1과 다르게 서버에 있는 Customer들의 Departure Time은 각기 다름
// 따라서 이들을 저장하여, 빨리 나가는 순서대로 뽑아내기 위하여 배열을 사용
```

5. 과제 수행 시행 오류

과제를 수행하는 도중 next_departure_time값을 S=C 인경우에 값을 정하기 위해서는 가게 내에서 서비스를 받고 있는 프로세스들 중에 가장 먼저 나가는 순으로 손님들을 관리를 해야했습니다. 따라서 처음에는 C++ STL라이브러리의 우선순위 큐로 구현을 하고자 하였으나 잘 되지 않아 배열이나 맵을 사용하여 하려고 인터넷을 조사하던 중 오른쪽의 깃허브 코드를 발견하였습니다. 이는 최초 제가 왼쪽의 열에서 하려고 했던 방법과 아이디어는 동일하고, 인덱스를 사용하는 방법이 더 편할 것으로 생각하여 해당 코드의 일부를 참고하였습니다.
(https://github.com/lukaswals/queuing-simulators/blob/master/mmc.c) 하지만 해당 코드에도 많은 에러가 있기에 함수만 참고를 하였습니다.
기존에 구성한 코드는 아이디어는 맞다고 생각은 하였는 데, STL에 대한 이해 부족으로 제대로 수행할 수 있도록 구성하지 못하였습니다.

기존에 구성한 코드 | **깃허브 참고하여 수정한 코드**

기존에 구성한 코드 (1)

```
/*
std::priority_queue<int, std::vector<int>, std::greater<
double fastest_departure = simulation_time;
double last_departure = 0.0;
double lastBusyTime;
*/
```

깃허브 참고하여 수정한 코드 (1)

```
int min_departure(double arr[], int capacity)
{
    int index = 0;

    for (int i=1; i < capacity; i++)
    {
        if (arr[i] < arr[index])
            index = i;
    }
//    printf(" - INDEX = %d \n", index);
    return index;
}

int idle_server(double arr[], int capacity)
{
    int index = 0;
    bool idle = false;

    for (int i=1; (i < capacity && !idle); i++)
    {
        if (arr[i] == SIM_TIME)
        {
            index = i;
            idle = true;
        }
    }
//    printf(" - INDEX = %d \n", index);
    return index;
}
```

기존에 구성한 코드 (2)

```
if (n==1)
{
    tb=elapsed_time;
    next_departure_time=elapsed_time+expntl(Ts);
    // tb : 손님이 1명일 때의 시간
    // 손님이 1명일 경우, 다음 도착시간은
    // 첫 손님이 서비스를 다 받고 난 다음의 시간
    // 각 손님 당 서비스 타임(Ts) 는 Exponential Function을 따름
    // 따라서 다음 도착시간 = 이전 프로세스(첫 프로세스) 도착 시간(0=elapsed_time) + Exponential(Ts)
    //   "" 그럼 왜 여기서는 Ta가 아닌 Ts를 사용하는 가????? ""
    //   n = 1 이 된 경우, 손님이 없다가 현재 손님이 다시 들어온 것.
    //   따라서 Base Time 다시 설정.
    // next_departure_time = 들어온 1개의 프로세스가 서비스를 받고 떠나는 시간
}
else if(n <= 10)
{
    next_departure_time = elapsed_time + expntl(Ts);
    if(next_departure_time < fastest_departure )
        fastest_departure = next_departure_time; // 가장 빨리 나가는 손님 등록
    if(next_departure_time > last_departure)
        last_departure = next_departure_time;
    Td.push(next_departure_time);
}
```

깃허브 참고하여 수정한 코드 (2)

```
if ( n <= 10)
{ // 서버가 커스터머를 받을 수 있다면
    arrayIndex = idle_server(custDepartures, 10);
    // 가능한 서버의 인덱스를 가져오고, 해당 인덱스에 현재 Customer의 Departure 시간 저장
    custDepartures[arrayIndex] = elapsed_time + expntl(Ts);
    // Customer가 떠나는 시간은 Mean Service Time의 Exponential함수를 통해 설정

    if ( n == 1)
    { // 손님이 0명에서 1명이 될 경우, 인덱스에 들어있는 값을 현재 프로세스의 Departure Ti
        nextDepartIndex = arrayIndex;
        next_departure_time = custDepartures[nextDepartIndex];
        lastBusyTime = elapsed_time; // 서버가 서비스를 시작하는 시간
    }
}
```

기존에 구성한 코드 (3)

```
if( n > 0 )
{
    if ( n < 10 )
    {
        next_departure_time = next_arrival_time+expntl(Ts);
        if(next_departure_time < fastest_departure )
            fastest_departure = next_departure_time; // 가장 빨리 나가는 손님 등록
        if(next_departure_time > last_departure)
            last_departure = next_departure_time;
        Td.push(next_departure_time);
    }

    if ( n >= 10 )
    {
        next_departure_time = fastest_departure + expntl(Ts);
        if(next_departure_time < fastest_departure )
            fastest_departure = next_departure_time; // 가장 빨리 나가는 손님 등록
        if(next_departure_time > last_departure)
            last_departure = next_departure_time;
        Td.push(next_departure_time);
    }
}
else
{
    next_departure_time = simulation_time;
    B+=elapsed_time - tb;
}
```

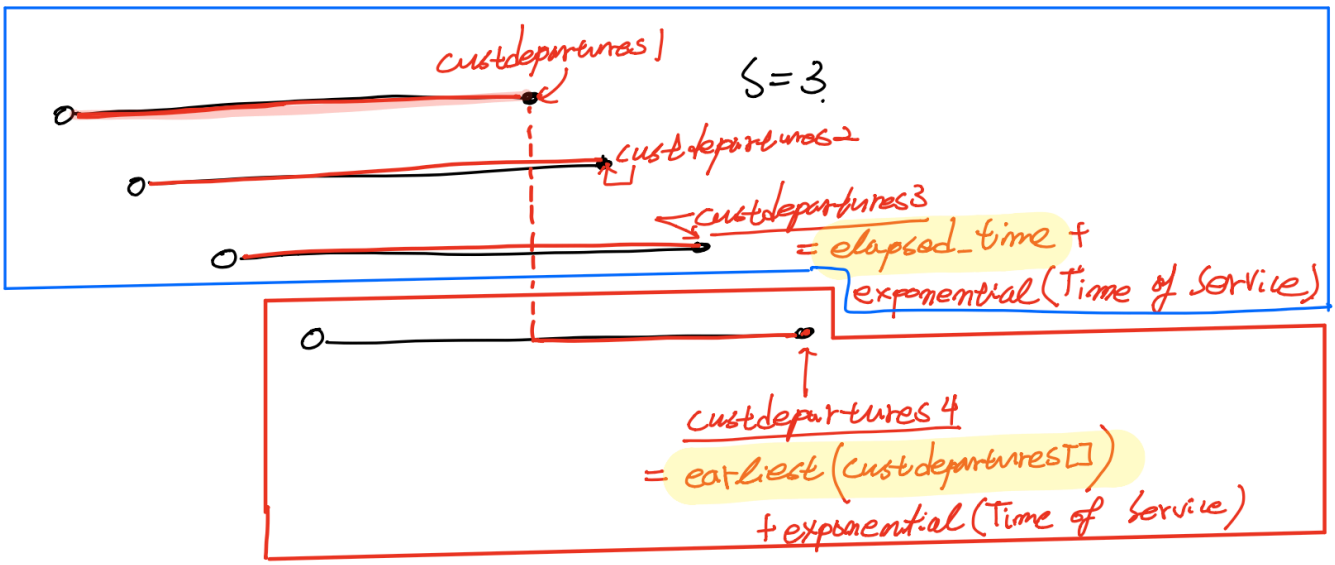
깃허브 참고하여 수정한 코드 (3) (이미지가 뒤집혀 인쇄됨)

```
if( n > 0 )
{
    if ( n >= 10)
        // 손님이 10개 이상이라면, 그 다음 손님이 떠나기 위해서는, "이전에 온 손님들 중 한 명이 서비스를 다 받은 뒤에" 서비스를 받을 수 있음
        // 따라서 이전 손님이 나간 현재시간 (elapsed_time)에 서비스를 받는 시간을 더함
        // 이 시간을 custDepartures 배열의 초기화된 인덱스에 저장
        custDepartures[nextDepartIndex] = elapsed_time + expntl(Ts);

    nextDepartIndex = min_departure(custDepartures, 10);
    // 배열에 저장된 시간 중 가장 빠른 Departure Time을 갖는 인덱스를 가져옴
    next_departure_time = custDepartures[nextDepartIndex];
    // 가장 빠른 이 된 손님을 통해 departure_time을 다음 손님이 나가는 시간으로 가져옴
}
else // 서버에서 이전에 들어온 마지막 손님이 떠남
{
    next_departure_time = simulation_time; // 변수 초기화
    B += elapsed_time - lastBusyTime;
    // 서버에 손님이 1명이 들어올 때부터 다음 때까지 서버는 사용중임.
    // 따라서 손님이 1명이 들어왔을 때 부터 0명이 될 때까지는 서버가 사용중인 시간.
}
```

6. next_departure_time 계산

아래의 예제는 M/M/S=3 일 경우, next_departure_time을 계산하는 방법입니다.
위에서 2개의 작성한 코드 모두 아래 그림처럼 S=C인 시스템에서 n > C 인 경우에 대해서 Next_departure_time을 계산하는 방법이 달라져야합니다. 이를 위해서 처음에 생각을 하였던 것은 우선순위 큐를 사용하여 진행을 하려 하였습니다. 왜냐하면 n > C인 경우에서 고려를 해야하는 것은 이미 시스템에서 서비스를 받고 있는 프로세스들 중 서비스를 빨리 받는 순으로 고려를 하여 관리를 하여야하는데 이를 위해서는 우선순위 큐가 효율적이라고 생각했기 때문입니다.
본인은 C++ STL의 priority queue를 사용하였지만 그 사용법을 정확히 숙지하지 못하여 결과에서 n의 값이 계속해서 음의 값으로 나왔습니다. 따라서 본인이 따로 priority queue 기능을 만들어서 해결하기 전에 인터넷에서 다른 사람들은 어떻게 처리하였는 지 조사를 하던 중, https://github.com/lukaswals/queuing-simulators/blob/master/mmc.c 에서 이를 인덱스 배열로 해결하는 것을 보았습니다. 이 깃에서 사용한 방법은 우선순위 큐의 작동과 유사하나, C개의 인덱스를 갖는 배열을 통해 해당 문제를 해결하는 것이였기에, 참고하여 프로젝트를 마무리 하였습니다.
해당 깃허브에 있는 코드는 초기 주어진 코드와 변수나, 동작 원리는 거의 동일하나, utilization을 구하는 방법이 잘못되어 있어, 본인의 코드에서 해당 값을 수정하여 코드의 주석에 상세히 설명하였습니다.



7. 가게를 효과적으로 운영하기 위해서 종합적인 정책이나 대책을 논리적으로 설명하시오

해당 이론 및 시뮬레이션을 통해 위에서 본 것과 같이, 손님이 많다 하여 테이블이나 종업원을 늘리는 것은 가게 운영의 효율성을 떨어뜨리게 됩니다. 또한 손님 1명당 가게를 방문하는 시간 간격과, 손님이 평균적으로 가게에서 음식을 먹고 떠나는 데 걸리는 시간들에 대한 평균값을 구하여야 합니다.

특히 여기서 λ(단위 시간당 손님 수)/μ(단위 시간당 서비스 처리 수) 의 비율이 1에 최대한 가깝게, 즉 단위 시간당 손님 수와 단위 시간당 서비스의 처리량(떠나는 손님의 수)를 최대한 비슷할 수 있도록 우동 로봇의 수를 배치하여야 가게의 효율을 높일 수 있습니다.

이러한 예로, 최근 골목식당에서 나온 필동 멸치국수 집, 떡볶이 집, 그리고 코너 스테이크 집을 보면 알 수 있습니다. 이 두 가게 모두 백종원의 골목식당에 반영된 이후 아주 많은 사람들이 가게 앞에서 진을 치며 기다리고 있었습니다. 그 이유는 영세한 가게이기에 테이블의 수(S)가 L(총 손님의 수)에 비해 턱없이 부족하였기 때문입니다. 하지만 그만큼 테이블들이 비어있는 시간(U)은 매우 적었습니다. 코너 스테이크 가게를 뺀 나머지 2개의 가게는 손님들의 수를 더 받기 위해서 테이블의 수(S)를 늘렸습니다. 하지만 그 결과는 손님들의 수는 계속적으로 비슷하나 빈 테이블의 수가 많이 늘게 되었고, 또한 이를 위해 다른 건물을 임대하였기에 결과적으로는 운영 효율성(U)만 떨어지게 된 것입니다.

따라서 효율적으로 가게를 운영하고 Profit을 만들기 위해서는, 가게에서 식사 중인 손님의 수 및 외부에서 대기하는 손님의 수에 대한 정확한 수치적인 분석이 필요하고, 이에 대해, 테이블(해당 과제에서는 우동 로봇)의 수에 따라 시뮬레이션을 통해 Throughput과 Utilization, 그리고 가게 운영비 이 3가지의 조건이 목표와 부합한 지에 대해 확인하여 운영을 할 것입니다.

- 소스코드 캡쳐 -

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>
#define norm_rand()  (rand()/(RAND_MAX + 1.0))

#define SIM_TIME 200000.0

double expntl(double);
int min_departure(double arr[], int capacity);// 인덱스를 이용하여, 가장 빨리 떠나는 Customer를 가져오기 위한 함수
int idle_server(double arr[], int capacity);// 서버의 Capacity 확인
double Ta=200.0, Ts=100.0;


int main() {
    int n;
    // Ta : Time of Arrival
    // next_arrival_tiem : 지난시간 + Exponential(Ta)
    // next
    // Ts : Time of service
    double simulation_time=SIM_TIME;
    double next_arrival_time, next_departure_time, elapsed_time;
    double B, C, L, s, tb, lastEventTime, W, X, U;
    n=0; // # of Customer
    next_arrival_time=0.0;
    next_departure_time=simulation_time;
    elapsed_time=0.0;
    B=s=0.0;
    C=0;
    lastEventTime=elapsed_time;
      double lastBusyTime = 0.0;
      int arrayIndex = 0;
      int nextDepartIndex = 0;
      double custDepartures[10];
      // MM1과 다르게 서버에 있는 Customer들의 Departure Time은 각기 다름
      // 따라서 이들을 저장하여, 빨리 나가는 순서대로 뽑아내기 위하여 배열을 사용

      for(int i = 0 ; i < 10 ; i++)
          custDepartures[i] = SIM_TIME; // Customer Departure Time 초기화

 /* Seed the random-number generator with current time so that the number will be different every
      time we run.*/
      srand((unsigned)time(NULL));

          while (elapsed_time < simulation_time)
          {
              if (next_arrival_time < next_departure_time )
              { /* event 1 : customer arrival */

                  elapsed_time=next_arrival_time; // New Customer가 도착한 시간
                  s = s + n*(elapsed_time-lastEventTime);
                  // 각 프로세스가 도착한 시간을 lastEventTime, 현재 프로세스가 도착하는 시간을
                      기준으로(elapsed_time)으로
                  // 그 시간 간격동안 있는 프로세스들의 수와 그 시간 간격을 더함으로써 서비스를 받고있는 n개의 프로세스가
                      서버에 있는 시간을 더함
                  n++; // 시스템 내의 customer 추가
                  lastEventTime = elapsed_time; // lastEventTime : 현재 Arrived Customer Time 등록
                  next_arrival_time = elapsed_time+expntl(Ta); // Mean Interarrival Time을
                      Exponential하여 현재 시간에 더해서 구함

                  if ( n <= 10)
                  { // 서버가 커스터머를 받을 수 있다면
                      arrayIndex = idle_server(custDepartures, 10);
                      // 가능한 서버의 인덱스를 가져오고, 해당 인덱스에 현재 Customer의 Departure 시간 저장
                      custDepartures[arrayIndex] = elapsed_time + expntl(Ts);
                      // Customer가 떠나는 시간은 Mean Service Time의 Exponential함수를 통해 설정

                      if ( n == 1)
                      { // 손님이 0명에서 1명이 될 경우, 인덱스에 들어있는 값을 현재 프로세스의 Departure Time으로
                          설정
                          nextDepartIndex = arrayIndex;
                          next_departure_time = custDepartures[nextDepartIndex];
                          lastBusyTime = elapsed_time; // 서버가 서비스를 시작하는 시간
                      }
                  }
              }
```

```
            else
            { /* event 2 : customer departure */
                // 서비스를 받고 있는 Customer 의 Departure Time보다 늦는 Arrival TIme을 갖는다는 것은
                // 서비스를 받고 있는 손님의 갯수가 1명이 줄어든다는 의미로, Cumstomer Departure.
                elapsed_time = next_departure_time; // 이전 프로세스의 떠나는 시간까지를 현재 기준 시간으로 잡음
                s+= n*(elapsed_time-lastEventTime);
                // 이전프로세스가 떠나는 시간부터 이전 프로세스가 도착한(떠난)시간 간격 동안, 서비스를 받고 있는 손님의 수를 그 시간간격을 곱합으로써
                // 서버내에서 서비스를 받고 있는 총 손님들의 시간을 더함
                n--;
                // 손님 떠남
                lastEventTime = elapsed_time; // 현재 손님 떠난 시간으로 기록
                C++;
                // 서비스를 받고 떠난 손님의 수 증가

                custDepartures[nextDepartIndex] = simulation_time; // 나간 손님 시간이 저장된 인덱스 초기화

                if( n > 0 )
                {
                    if ( n >= 10)
                        // 손님이 10개 이상이라면, 그 다음 손님이 떠나기 위해서는, "이전에 온 손님들 중 한 명이 서비스를 다 받은 뒤에" 서비스를 받을 수 있음
                        // 따라서 이전 손님이 나간 현재시간 (elapsed_time)에 서비스를 받는 시간을 더함
                        // 이 시간을 custDepartures 배열의 초기화된 인덱스에 저장
                        custDepartures[nextDepartIndex] = elapsed_time + expntl(Ts);

                    nextDepartIndex = min_departure(custDepartures, 10);
                    // 배열에 저장된 시간 중 가장 빠른 Departure Time을 갖는 인덱스를 가져옴
                    next_departure_time = custDepartures[nextDepartIndex];
                    // 가장 빠른 이 전 손님들 중의 departure_time을 다음 손님이 나가는 시간으로 가져옴
                }
                else // 서버에서 이전에 들어온 마지막 손님이 떠남
                {
                    next_departure_time = simulation_time; // 변수 초기화
                    B += elapsed_time - lastBusyTime;
                    // 서버에 손님이 1명이 들어올 때부터 나갈 때까지 서버는 사용중임.
                    // 따라서 손님이 1명이 들어왔을 때 부터 0명이 될 때까지는 서버가 사용중인 시간.
                }
            }
            printf("n : %d  lastBusyTime : %f   lastEventTime : %f\n", n, lastBusyTime, lastEventTime);
        }

    X = C / elapsed_time;
    printf("throughput = %f\n", X);
    // elapsed time = T

    U = B / elapsed_time;
    printf("utilization = %f\n", U);

    L = s / elapsed_time;
    printf("mean customer no. in store = %f\n", L);

    W = L/X;
    printf("mean \"Dwell\" time per customer = %f\n", W);

    printf("L : %f  W : %f \n", L, W);


}

double expntl(double x)
{   /* 'expntl' returns a psuedo-random variate from a negative exponential distribution with
    mean x */
    return(-x*log(norm_rand()));
}

/**************************************************************************
*    min_departure(double arr[], int capacity)
**************************************************************************
* Function that return the index of the minimum departure time
* - Input: arr (array of departures)
* - Input: capacity (size of the array)
**************************************************************************/

int min_departure(double arr[], int capacity)
{
    int index = 0;

    for (int i=1; i < capacity; i++)
    {
        if (arr[i] < arr[index])
            index = i;
    }
//    printf(" - INDEX = %d \n", index);
    return index;
}
```

< 깃허브 참고한 부분입니다. >

```
/**************************************************************************
*     min_departure(double arr[], int capacity)
**************************************************************************
* Function that return the index of the minimum departure time
* - Input: arr (array of departures)
* - Input: capacity (size of the array)
**************************************************************************/

int min_departure(double arr[], int capacity)
{
    int index = 0;

    for (int i=1; i < capacity; i++)
    {
        if (arr[i] < arr[index])
            index = i;
    }
//    printf(" - INDEX = %d \n", index);
    return index;
}


/**************************************************************************
*     idle_server(double arr[], int capacity)
**************************************************************************
* Function that return the index of an "idle server"
* It's used to determine to which position of the array we will save the
* departure time of the client (which server is serving the customer)
* - Input: arr (array of departures)
* - Input: capacity (size of the array)
**************************************************************************/
int idle_server(double arr[], int capacity)
{
    int index = 0;
    bool idle = false;

    for (int i=1; (i < capacity && !idle); i++)
    {
        if (arr[i] == SIM_TIME)
        {
            index = i;
            idle = true;
        }
    }
//    printf(" - INDEX = %d \n", index);
    return index;
}
/*
 식당에 들어와서 식사 중인 손님 10명 중, 가장 빨리 먹고 나갈 예정인 사람 다음의 사람을 처리하는 방법으로
 1. Priority Queue를 사용하려고 해보았으나, 잘 되지 않아
 2. 인덱스나 맵을 사용해보려 했습니다
 이 부분을 제출 기간 내에 해결을 하지 못하여 아래의 깃허브에서 해당 부분을 참고하여 사용하였습니다.
 전부 다 카피를 한 것은 아니고, 인덱스 부분만 아이디어를 가져왔고
 의미에 맡게 변수들을 재설정하여 작성하였습니다.

 가장 중요한 점 : 서버가 가득 차있을 때와 서버에 가용 공간이 있을 때의 손님이 떠날 시간을 예측하는 것이 중요합니다.
 이는 서버가 1개일 때와 여러 개 일 때 다름
 https://github.com/lukaswals/queuing-simulators/blob/master/mmc.c
 */

```